Lexical Analyzer

Igor Figueira Pinheiro da Silva - 15/0129921

University of Brasília

1 Motivation

The work presented here was developed for the Compilers class from University of Brasília. All students must develop a compiler for a simplified version of the C programming language with some small changes. The development is divided in 5 steps: lexical analysis, syntatic analysis, semantic analysis, intermediate code generation, and work presentation.

2 Lexical Analysis Description

We introduce the lexical analyzer, which receives the source code as input and outputs a stream of all tokens and its identifiers. Errors are also checked and are identified accordingly. First of all, we define regular expressions to identify all the basic structures of the defined language. Structures such as comments and whitespace characters must be ignored. Data types, operators, flow control symbols, and other important symbols have regular expressions specified to identify their type and extract the data correctly.

2.1 Data Structures and Functions

Column number and line number are saved in order to report unrecognized tokens. These values are changed using the *update_position* function. No data structures are used until this moment because currently the lexical analyzer only logs all the tokens that are recognized or return an error. In the future data structures such as the symbol table will need to be implemented.

3 Test Files

Examples 1 and 2 are correct according to the lexical analysis. Example 3 has an error on line 5, in which num1 is assigned the unknown symbol "#". Example 4 has a missing the closing double quote on line 2.

3.1 Example 1

```
int age;

int main() {
  writeln("Enter your age: ");
  read(age);

  if(age < 50){
    writeln("Sorry! Try again next year :(");
  } else {
    writeln("Congratulations! You finally did it!");
  }
  return 0;
}
```

3.2 Example 2

```
float num1;
float num2;

int main() {
  num1 = 1.5;
  num2 = 1.4;

  writeln(num1 + num2);
  return 0;
}
```

3.3 Example 3

```
int num1;
int num2;

int main() {
  num1 = #;
  num2 = 1.4;

  writeln(num1 + num2);
  return 0;
}
```

3.4 Example 4

```
int main() {
  writeln("missing double quote);
}
```

4 Compiling instructions

In order to compile and run the lexical analyzer, open a bash terminal and run the following command:

```
$ flex lex_analyzer.l && \
  gcc lex.yy.c -o lex && \
  ./lex < your_example.c
```

References

1. Aho, A., Lam, M., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, & Tools. 2nd edn. Pearson, Boston (2007)
2. Levine, J.: Flex & Bison. 2nd edn. O'Reilly, California (2009)

Attachment 1 - Context Free Grammar

```
<translation-unit> ::= {<external-declaration>}*

<external-declaration> ::= <function-definition>
                         | <declaration>

<function-definition> ::= {<declaration-specifier>}* <declarator> {<declaration>}* <compound-statement>

<declaration-specifier> ::= <type-specifier>

<type-specifier> ::= int
                   | float
                   | double
                   | set
                   | elem

<specifier-qualifier> ::= <type-specifier>

<logical-or-expression> ::= <logical-and-expression>
                          | <logical-or-expression> || <logical-and-expression>

<logical-and-expression> ::= <inclusive-or-expression>
                           | <logical-and-expression> && <inclusive-or-expression>

<inclusive-or-expression> ::= <exclusive-or-expression>

<exclusive-or-expression> ::= <and-expression>

<and-expression> ::= <equality-expression>
                   | <and-expression> & <equality-expression>

<equality-expression> ::= <relational-expression>
                        | <equality-expression> == <relational-expression>
                        | <equality-expression> != <relational-expression>

<relational-expression> ::= <shift-expression>
                          | <relational-expression> < <shift-expression>
                          | <relational-expression> > <shift-expression>
                          | <relational-expression> <= <shift-expression>
                          | <relational-expression> >= <shift-expression>

<shift-expression> ::= <additive-expression>

<additive-expression> ::= <multiplicative-expression>
                        | <additive-expression> + <multiplicative-expression>
                        | <additive-expression> - <multiplicative-expression>

<multiplicative-expression> ::= <cast-expression>
                              | <multiplicative-expression> * <cast-expression>
                              | <multiplicative-expression> / <cast-expression>
                              | <multiplicative-expression> % <cast-expression>

<cast-expression> ::= <unary-expression>
```

```
<unary-expression> ::= <postfix-expression>
                     | ++ <unary-expression>
                     | -- <unary-expression>

<postfix-expression> ::= <postfix-expression> ( {<assignment-expression>}* )
                       | <postfix-expression> ++
                       | <postfix-expression> --

<constant> ::= <integer-constant>
             | <character-constant>
             | <floating-constant>

<expression> ::= <assignment-expression>
               | <expression> , <assignment-expression>

<assignment-expression> ::= <unary-expression> <assignment-operator> <assignment-expression>

<assignment-operator> ::= =
                        | *=
                        | /=
                        | +=
                        | -=

<unary-operator> ::= +
                   | -
                   | !

<type-name> ::= {<specifier-qualifier>}+

<parameter-type-list> ::= <parameter-list>
                        | <parameter-list> , ...

<parameter-list> ::= <parameter-declaration>
                   | <parameter-list> , <parameter-declaration>

<parameter-declaration> ::= {<declaration-specifier>}+ <declarator>
                          | {<declaration-specifier>}+

<declaration> ::=  {<declaration-specifier>}+ {<init-declarator>}* ;

<init-declarator> ::= <declarator>
                    | <declarator> = <initializer>

<initializer> ::= <assignment-expression>
                | { <initializer-list> }
                | { <initializer-list> , }

<initializer-list> ::= <initializer>
                     | <initializer-list> , <initializer>

<compound-statement> ::= { {<declaration>}* {<statement>}* }

<statement> ::= <expression-statement>
```

```
              | <compound-statement>
              | <selection-statement>
              | <iteration-statement>
              | <jump-statement>

<expression-statement> ::= {<expression>}? ;

<selection-statement> ::= if ( <expression> ) <statement>
                        | if ( <expression> ) <statement> else <statement>

<iteration-statement> ::= for ( {<expression>}? ; {<expression>}? ; {<expression>}? ) <statement>

<jump-statement> ::= return {<expression>}? ;
```